0S/(01-MP13   [illegible]

Automatic Data Traffic Control on DSM Architecture

Michael Frumkin*, Haoqiang Jin* and Jerry Yan†

NAS Systems Division
NASA Ames Research Center

Abstract

We study data traffic on distributed shared memory machines and conclude that data placement and grouping improve performance of scientific codes. We present several methods which user can employ to improve data traffic in his code. We report on implementation of a tool which detects the code fragments causing data congestions and advises user on improvements of data routing in these fragments. The capabilities of the tool include: deduction of data alignment and affinity from the source code; detection of the code constructs having abnormally high cache or TLB misses; generation of data placement constructs. We demonstrate the capabilities of the tool on experiments with NAS parallel benchmarks and with a simple CFD application ARC3D.

1. Introduction

Distributed shared memory (DSM) machines are proving to be an efficient solution for computationally intensive scientific calculations because of ease of writing parallel programs and gaining speedup of the parallel code. DSM machines allow easy to use parallel programming paradigms based on OpenMP and on Java Threads. The main advantage of shared memory architecture is providing user with a global address space for all threads executed across the machine. The threads have logically equal access to the application data regardless of the physical mapping of the memory allocated to the application. DSM machines employ at least two levels of cache memory and an additional hardware such as TLB (translation-look-aside buffer) and Directories to track the location of data been processed, to keep the data in a coherent state, and to hide the latency of memory access due to the physical distribution of memory.

While DSM hardware and its operating system liberate user from the duty of the explicit data and computation location management, a poor data distribution can result in significant increase of data traffic, which causes data congestions and loss of performance. Some well known data congestion problems include: excessive data cache misses, false sharing and excessive TLB misses. The methods of improving the data traffic on distributed memory (DM) machines involve nests reordering, data partitioning, computation and communication overlapping [16][25][26][27]. An engineering of the high performance of parallel code on DSM machines, as on DM machines, requires assignment of well balanced load to each thread and organization of congestion free data traffic [24]. The methods for improving data traffic include data placement, data grouping, and data transposition. Improvements in data locality usually are translated into increase in per-

*Computer Sciences Corporation. M/S T27A-2, NASA Ames Research Center, Moffett Field, CA 94035-1000; e-mail: {frumkin,hjin}@nas.nasa.gov

†NAS Division, T27A-2, NASA Ames Research Center, Moffett Field, CA 94035-1000; e-mail: yan@nas.nasa.gov

formance of serial code and into better scalability of parallel code both on DM and on DSM architectures. In some cases the compilers are able to improve data traffic by data prefetching, by overlapping the computations with memory access, and improve cache performance by nest tiling.

In the first part of this paper we demonstrate effects of improving data traffic on performance of the applications running on DSM machines. We show that the data traffic depends on two components: the data-to-computations affinity and the data-to-data affinity. The affinity relations can be obtained as a result of program analysis and than can be used by a tool for data traffic improvements on DSM machines. We illustrate our findings on running on array-oriented scientific codes written in FORTRAN and running on SGI Origin2000. The codes are designed to solve partial differential equations on structured grids and usually on each iteration perform few operations per datum and, as a result, have a tendency to be memory bound.

In the second part of this paper we show that many data traffic problems can be identified in the code automatically and the user can be advised by a tool on possible data congestions and ways to resolve them. The tool extends the functionality of ADAPT (Automatic Data Alignment and Placement Tool) [9] which analyzes data affinity and generates HPF stile directives. The tool also employs information provided by CAPO [11] which annotates code with OpenMP directives.

Our experiments are accomplished on SGI Origin 2000 24 processor machine under IRIX64 6.5 and MIPSpro f77 compiler, version 7.3.1.1m. We use two codes to illustrate data traffic effects in DSM machine. The first code is SP of NAS Parallel Benchmark set [4] (both, FORTRAN serial and OpenMP versions), which uses an ADI method to solve the Navier-Stokes equation on a cubical grid [23]. The second code is a simple nest for computation of an explicit stencil operator $q=Ku$, where u and q are two three-dimensional arrays of the dimensions nx, ny and nz, and K is a 13 point star operator with constant coefficients, see Figure 1. Finally, we demonstrate the capabilities of the tool on experiments with NAS parallel benchmarks and with a simple CFD application ARC3D.

2. Data-to-Computations Affinity

In order to reduce processor-memory performance gap [15] a number of data traffic control techniques have been developed including vectorization, pipelineing, prefetching, tiling and blocking [3][16][17][25][26]. An appropriate application of these techniques is based on analysis of data-to-computations and data-to-data affinity. Informally a datum is affine to an instruction if it is used/produced by the instruction (see Section 4 for a formal definition).

We work with parallel programs where the thread number is known for each instruction. This can be an OpenMP program with directives specifying the thread which will be processing each instruction[*], an MPI or HPF program where a symbolic number of the processor executing each instruction is known. For each datum we define the usability vector be a vector with component i equal to the number of instructions affine to the datum executed by the thread i. The index of the maximal component of the usability vector can be used to specify the thread affine to the datum. For good data locality the data affine

[*] In OpenMP programs with dynamic thread scheduling the actual thread executing instruction can vary. So here we assume a static scheduling of OpenMP programs.

to a particular thread should be grouped together and pages having the highest affinity to a particular thread should be allocated in the memory of the processor where the thread is running. In the computations involving arrays the user can control both array shape (by padding dimensions) and placement of individual pages allocated for array (by controlling the page placement policy and the page size). The page placement depends on the application placement policy and on the thread which writes to the page first, [24].

For illustration of data-to-computations affinity let us consider computations of an explicit operator $q=Ku$, Figure 1. The profile of the memory access for `kji`, `kij` and `jik` nest ordering obtained with SGI `dprof` tool is shown on Figure 2.The reason for high memory access interference in the `jik` ordering is that each thread uses few data from each memory page allocated to the arrays. The large stride in the memory access also causes a significant number of TLB misses. A technique for alleviating this problem was developed in HPF compilation systems [3] and called data transposition. It is based on introduction of a separate array with dimensions i and k transposed, copying the original data into the transposed array, performing computations with the transposed array, and copying results back to the original array. The transposition dramatically reduces the interference of threads, see Figure 3. The transposition method is relatively expensive and improves performance only if the interference volume is large enough to amortize the transposition cost.

An interference of the threads reading data from the same page can cause a bottleneck in the channel of the memory hosting the page. If two or more threads running on different processors write data so closely located in the main memory that they are mapped into the same line of the secondary or primary cache then the performance loss even more significant. In this case each write results in an invalidation of the cache line in all threads accessing it, causing effect known as false sharing.

Tables 1 and 2 show the performance, the primary data cache (PDC) misses, the secondary data cache (SDC) misses and the TLB misses in the nest. The nest with `kji` loop ordering has the best performance. The performance can be further improved by allocation of the pages affine to each processor in the memory local to it by touching the pages in an appropriate order. The last lines in both tables show performance of the code with appropriately parallelized data initialization nest. The page placement can be profiled with another SGI tool called `dlook`. It shows that as a result of initial data placement the pages allocated to the arrays are distributed across the processors.

A reduction in TLB misses can be also achieved by increasing of the page size. If the page size, however, is too big then an array can be concentrated on few processors creating a bottleneck for accessing it by threads executed on other processors. The preferable page size depends on the number of processors involved and for given array dimensions it should be chosen as the total array size divided by the number of threads accessing it. For example, using page size 1 MB and 256K (instead of default 64K) improves performance of SP benchmark, class A, by 10% on 2 and 4 processors respectively (typical array size in this benchmark is 64^3 of 8 byte words=2MB).

3. Data-to-Data Affinity

Two datums are affine if both are used at the same instruction executed during the program run. A grouping of affine data items together and organizing groups into a continuos stream often improves the program performance by hiding the memory latency. In general the affinity relation is a many-to-many relation and, as a result, there are many ways to group affine data items. In [7] it is shown that the possibility of grouping affine array elements is significantly affected by the geometry of the self interference lattice of the array. The self interference lattice can be defined as a lattice of array elements mapped into the same word in the cache [7], or, equivalently, as a set of solutions of the Cache Miss Equation [10].

We illustrate the grouping of affine elements and the memory access interference on example of computation of a stencil operator $q = Ku$, Figure 1. Let us choose arbitrary time stamps t_i, $i=1,...,n$, with t_1 and t_n respectively been start time and end time of the computation. Let Q_i be the set of q elements computed at time interval $[t_i,t_{i+1}]$, $i=1,...,n-1$, and let U_i=aff(Q_i) be the set of u elements affine with the elements of Q_i, see Figure 4. All elements of U_i are accessed during computation of Q_i. The number of cache misses during computation of Q_j can be reduced if we would be able to keep elements of all U_{ij}, $i<j$ in the cache (here $U_{ij} = U_i \cap U_j$). If, however, the number of elements in all U_{ij} ($i<j$) exceeds the cache size, then at least $m_j = \sum_{i,i<j} |U_{ij}| - S$ (here S is the cache size) must be dropped from the cache and reloaded later resulting in m_j misses (this type of misses called *replacement miss* [10]). On the other hand, if each U_i fits into the cache then the total number of replacement misses will be at most $\sum_i \sum_j |U_{ij}|$.

The following *non interference conditions* is necessary for computations of Q_i without replacement misses be possible (below A is the cache associativity number):

- no q-u cross interference: for any element x of Q_i no more than $A-1$ elements of aff(x) can be mapped into the same cache line as x
- no u self interference (and, symmetrically, no q self interference): for any element x of Q_i no more than A elements of aff(x) can be mapped into the same cache line

Both self and cross interference cause the effect known as cache trashing [24]. It results in a significant increase of cache invalidation and cache misses. On the other hand, if neither of interferences occurs then we can construct a covering of the iteration space with interference free sets in the following way. We pick an element q_1, choose Q_1 to be an interference free neighborhood of q_1[*]. Let $L=\{q_i\}$ be interference lattice of q, that is the set of elements of q mapped onto the same word in cache as q_1 and Q_i be a translation of Q_1 by a vector of q_i-q_1. If $\{Q_i\}$ covers all elements of q then we have a good tiling of the iteration space. Otherwise we choose an array element not covered by $\{Q_i\}$ and repeat the tiling operation.

The interference free neighborhood can be chosen in a number of ways. Three interference free sets are shown in Figure 5. In [7] it is shown that a fundamental parallelepiped of a reduced basis of the self interference lattice in many cases provides an optimum covering. Figure 6 shows cache misses for two methods for calculation of the second order stencil: 1) compiler optimized (the compiler flags are described below) and 2) tiled with a successive minima parallelepiped [5].

To illustrate the primary and secondary data cache effects in DSM machine we consider a calculation of 4D array `lhs` in `lhsx`, `lhsy` and `lhsz` (see nest 1 of SP, serial version). For computation of the values of `lhs` on a k-plane in `lhsz` the values of two arrays

[*]We say that two array elements at a distance 1 if they are in the affinity relation. A neighborhood of an element q_1 is a set of all elements of q at some distance d from q_1.

on the planes *k-1*, *k*, *k+1* are used. The computations are optimized in such a way that the intermediate values along each *k*-line are computed and stored in temporary 1D arrays. For the computations of the final values of `lhs` the two temporary arrays are used, see `lhsz` nest in Figure 7. The appropriate nests in `lhsx` and `lhsy` are organized similarly having *k* index interchanged with *i* and *j* respectively.

We have compiled code with MIPSpro f77 compiler, using the flag `-O3 -NLO:prefetch=0`. The last flag prevents the compiler from doing an aggressive prefetching to reduce time processor stalls waiting for data. We profiled these nests for class A (*nx=ny=nz=64*) using a hardware counters profiling library*. In the benchmark the arrays were padded to make the first and second dimensions of all arrays equal to 65. The measurements presented in the columns 2 through 4 of Table 3 show that the PDC misses are about a factor 2.6 larger for `lhsy` and `lhsz` than for `lhsx`. This is a consequence of little reuse of cache lines loaded with each cache miss. The total number of referenced data in the nest is about 14 MB (7 arrays of the size 64^3 words or 2 MB each). The total size of the secondary cache is 4 MB and each cache load loads whole line of 128 B. So, in each case it corresponds to about $22*10^5*128$ B ~ 280 MB of loaded (or written back) data.

A significant increase in the number of TLB misses for `lhsz` is well in line with the fact that this nest accesses the data dispersed across many memory pages (this effect similar to the large number of TLB misses in `jik` line of Tables 1 and 2). The increase in the execution time for the `lhsy` relative to `lhsx` is accounted for the larger number of PDC misses. For `lhsz` increase in the execution time is accounted mostly for the TLB misses.

In our optimization we removed temporary arrays by sacrificing calculations of common expressions with the `SimpleFunction()`. Then we removed the two internal loops and performed calculation in `kji` order.This transformation increases the number of executed floating point instructions, however, it allows to decrease the number of TLB and PDC misses. The resulting nest `lhsz_r` is shown in Figure 7. The measured events for the optimized nests (the last 2 columns of Table 3) show as decrease in the PDC misses and TLB as an improvement in the execution time. The number of PDC misses in `lhsz_r` is still bigger than the number of PDC misses in `lhsx` which can be explained by incomplete reuse of data from *k-1* and *k+1* planes. The reduction of the execution time in spite of increasing of the number of the floating point instructions in `lhsy_r` and `lhsz_r` indicates that the original nests are memory rather than CPU bound.

4. Automation of Data Traffic Control

The methods of data grouping, placement and localization described in the previous sections can be formalized and implemented in an automatic tool assisting the user in improvement of data traffic in her application. We implemented these methods in a tool which is able to analyze the code and detect constructs which potentially can cause data congestions and loss of application performance. We used ADAPT (Automatic Data Alignment and Placement Tool) [9], as a basis for the implementation. ADAPT uses CAPTools [12][13] generated application data base, extracts data affinity, deduces data distributions and produces a code annotated with the HPF [17] directives. In order to

*The hardware counters library allows to compute a histogram of 32 hardware events on SGI machines using R10000 and R12000 processors. The library uses `ioctl` system call provided by IRIX64 6.5 to access the counters. The library was developed by Amy Fornal.

incorporate the parallelization information ADAPT takes additional input from the CAPO tool which annotates code with OpenMP directives [11]. The input consists of a list of parallelized loops augmented with a lists of private/shared variables for the loops. Our extensions to ADAPT includes the following functionality:

- detection of application sources -> initial data placements
- detection of high number of TLB misses -> loop interchange/array transposition
- detection of self and cross interference in loop nests -> array padding and offsets
- generation of interference free tiles

These features of ADAPT are based on analyzing the data-to-computations affinity and the data-to-data affinity. The case when affinity can be explicitly represented by affine mappings was considered in [20]. In many codes, however, the mappings are nonlinear or are many-to-many mappings and more general technique based on calculation of affinity relations should be applied to such codes.

Data-to-computations affinity. We represent program by a bipartite graph called program affinity graph. Let C be the set of instructions of a program P, i.e. a set of program statements executed during the program run, and let D be the program data, i.e. the set of memory locations referenced during the program run. We say a memory location d (a datum) is affine to an instruction c if the value at address d is either operand or result of c. The program affinity graph has C and D as the vertices of the parts and an arc connecting each instruction with data affine to it. The direction of an arc connecting d and c is toward c or away from c depending on whether d is operand or result of c (it is possible that several arcs connect a datum and an instruction). Many program properties can be expressed in terms of its affinity graph. For example, an instruction c_2 depends on an instruction c_1 if there is a direct path from c_1 to c_2. Otherwise, if c_1 and c_2 correspond to the program statements located in the same basic block [1], then c_1 and c_2 are independent and can be executed in any order.

It is unpractical to use the full affinity graph for calculation of the usability vector (see Section 2) since the size of the graph (order of the number instructions executed by a program) usually is too big. The analysis of the graph can be simplified by indexing the instructions belonging to the same nest and the memory locations belonging to the same array. In this case the arcs connecting data and instructions can be expressed as a pair of expressions `(I;idx(I))` where `I` is a vector loop index and `idx(I)` is a memory address of an array element referenced at iteration `I`, see Figure 7. In most of the cases in our application domain (Partial Differential Equations on structured grids) the index function is linear function of `I` with symbolic coefficients known at compile time. The are few nests in our applications where this is not a case. These nests include the core of the FFT algorithm where the `idx`$(i,j,k)= i+j*2^k$ for `kji` loop nest; nests working with multiple grids where `idx` function is read from a file; nests working with specially enumerated grid points use `idx` function stored in a precomputed array. The tool indicates the nests with nonlinear access functions without any further analysis of the nests. In most nests with the linear access function the coefficients of the matrix representing the `idx` function are elements of the set {-1,0,1}, with an exclusion are the multigrid methods where the coefficients are multiple of 2.

Some properties of the program can be deduced using only symbolic information on the coefficients of `idx` function (see the thread noninterference condition below) others require knowledge of the actual numerical values of the coefficients (see the subsection on generation of interference free tiles). If the property of the program can be expressed in a symbolic form but can't be verified without knowing the numerical values of the coefficients the tool inserts some performance warning constructs in the code and the user obtains the warning in runtime.

Data-to-data affinity relation. For a pair of arrays used in the same loop nest statement, we define the affinity relation as a correspondence between array elements referred with the same value of the loop index. The affinity relation can be represented as a list of pairs*:

```
          do I from PI
            q(idxq(I))=u(idxu(I))
          end do
c         Aff(q,u)={(idxq(I);idxu(I)), I from PI}
```

The affinity relation can be deduced for each pair of arrays in each nest statement. A control dependence results in affinity relations between the arrays involved in the control statement and all arrays in each basic block immediately dominated by the statement. The most common case we observe in our applications is one-to-few affinity relations between arrays resulted from difference operators on structured discretization grids. These relations can be approximated by a stencil (i.e. by a set of vectors with constant elements) and we call them stencil relations [9].

The chain rule allows to deduce the affinity for arrays used in different statements of the same nest:

```
           do I from PI
             u(idxu1(I))=s(idxs(I))
             q(idxq(I))=u(idxu2(I))
           end do
c     Aff(q,s)={(idxq(I);idxs(J)),
c                         J=max{j: j<=I,idxu1(j)=idxu2(I)}}
```

where the max operation and inequality $\{j<=I\}$ are performed in the lexicographical order imposed by the nest indices. For statements with different nesting the chain rule is similar, see [9]. The chain rule allows to construct an affinity relation along each directed path in the nest data flow graph passing only through privatizable variables. The union of these relations over all directed paths to q from u forms the nest affinity relation between q and u. The relation lists all elements of u used for computation of the element of q and can considered as one-to-many mapping.

Checking Thread Noninterference. This condition can be formulated as nonoverlapping

*We use multidimensional indices, functions and domains in this section. It makes the presentation more compact and the analogy between loops and nests more transparent. For example, instead of

```
do i=1,nx
  do j=1,ny(i)
    do k=1,nz(i,j)
      q(idxq1(i,j,k),idxq2(i,j,k),idxq3(i,j,k))=
             u(idxu1(i,j,k),idxu2(i,j,k),idxu3(i,j,k))
    end do
  end do
end do
```

we write

```
do I from PI
  q(idxq(I))=u(idxu(I))
end do
```

of the address spaces accessed by different threads: if a thread accesses array elements at addresses A and B then no other thread accesses an array element at address C, if A<=C<=B. In particular, it means that the interference shown on the bottom (`jik`) insert in Figure 2, does not happen. If the noninterference condition is satisfied then the memory accessed by a thread can be placed at the memory of the processor running the thread, improving the data locality.

Consider a single nest of a parallel program and assume that the parallelized loop is known. If an array access function represented as a linear function of the nest indices *i,j,k* with symbolic coefficients *a,b,c*:

$$addr_p(i, j, k) = ai + bj + ck + cwp$$

where, *p* is the thread number, *w* is the number of iterations per thread, *0<=i<nx, 0<=j<ny, 0<=k<w, 0<=p<NUM_THREADS*. In this case the necessary and sufficient condition for thread noninterference can be formulated as

$$c > a(nx - 1) + b(ny - 1)$$

If there are multiple array access functions per array then we check the noninterference condition for each function. This, however, is not sufficient for thread noninterference. For example, in stencil type computations the access functions differ by a constant term (independent on *i,j,k*) and cause small thread interference, see interference between consecutive threads in the first two inserts of Figure 2.

Detection of High TLB misses. Table 1 indicates that the large number of TLB misses results from large memory stride due to iterations of the innermost loop of the nest. Our TLB miss test checks two conditions:

- the number of iterations of the innermost nest exceeds the TLB_SIZE
- the distance between the first an last address accessed in the innermost loop exceeds the PAGE_SIZE*TLB_SIZE

If both conditions can be proved to be true then the user gets a warning about high TLB misses in the nest. Otherwise, if both conditions can't be proved to be false then the tool inserts a runtime check in the code.

Checking Cache Unfriendly Access Patterns. In general, cache friendly computations involve good temporal and spacial locality [15] and can not be expressed in simple terms [10]. However, some necessary conditions for cache friendly computations can be formulated and checked. The first condition is simple: the coefficient at the innermost loop index is 1. Otherwise, nonunit stride in memory access can cause underutilization of data loaded into the cache. The other two conditions, self and cross array interference are formulated below.

Detection of Self Interference. If the self affinity relation can be expressed in the form stencil vectors the tool represents the addresses of the corresponding array elements as a polylinear functions of array sizes and the index coefficients. Then for each pair of the stencil vectors after common terms elimination in the addresses differences it generates a set of constraints for the array dimensions in the form *nx*ny != k*S*, where *nx,ny* are the array sizes, *S* is a cache size and *k* is a small integer. If *nx* or *ny* is not known at compile time a test for a satisfiability* of these inequalities is inserted in a program and in run time

*Note that if array sizes are known the satisfiability can be checked in liner time on the system size.

user gets a warning about possible self interference.

Detection of Cross Interference. The cross interference between two arrays happens when affine elements of the two arrays are mapped to the same cache location. Checking of the cross interference is similar to the checking of self interference with a difference that it involves the inter array offset and dimensions of both arrays.The cross interference constraints are represented by a polylinear equality as self interference: *nxa*nya+off_a_b+nxb*nyb!=k*S, k=1,2,3.*

Detection of the Data Sources and the Initial Data Placement. On some DSM machines memory pages are allocated on the processor which touches the page first, hence for initial data placement in an application we detect the constructs where data are initialized. We found that all data initialization constructs in our codes have one of following types:

- reading data from a file
- receiving data from another process
- initialization of arrays from another array
- initialization of an array with an intrinsic function (such as random number generator)

These constructs are easily detectable by our tool and a data placement directive (in the form of HPF ALIGN, DISTRIBUTE directives) is issued before each construct.

Generation the Interference Free Tiles. For generation of the interference free tiles we construct a successive minima parallelepiped, see Figure 5. In general case the parallelepiped sizes are discrete functions of the array dimensions and can be computed only for the arrays with known dimensions. Our algorithm finds the successive minima by a blowing a cube with the center in the origin and freezing a face as soon as it attains a lattice point. The parallelepiped is final if each its face contains a lattice point. An improvement in the cache misses by tiling of a nest of a second order explicit operator on a 3D grid with the successive minima parallelepipeds is shown in Figure 6.

5. Experimental results

We applied the tool to SP, BT and LU of NAS Parallel Benchmarks [4] (optimized OpenMP version PBN-O) and to an aerodynamic code ARC3D [8]. The tool was able to generate a file containing advising information on:

- nests for initial data placements
- nests with nonunit strides and possible loop interchange
- nests with big strides and possible data transpositions
- nests with self or cross interference
- tiles for improving cache performance

We analyzed the advising file for SP Benchmark and inserted the appropriate changes into code by hand. The performance results are shown in Figure 8. The tool detected 4 nests where data were initialized. The nests with nonunit strides were detected in `rhs`, `zsolve`, and `exact_rhs`. A nest with a big stride was detected in `zsolve`. No self interference was detected (the padding of the second and third dimensions in the benchmark was sufficient). The cross interference condition was presented in the form that the array offsets can not be equal to a multiple of cache size plus a stencil vector offset (the address of the array element represented by the vector). Two types of tiles were generated: (8,2,8) for the nests working with (65,65,64) arrays; and (5,8,2,8) for nests working with (5,65,65,64) arrays.

The initialization nests were properly parallelized in the original program, so no additional data placements were needed. In Figure 8 we showed the effect of a concentration of the arrays in the memory of a single node by introducing a nest for touching data by the master thread (totoal_no_placement curve). We interchanged the loops in the nest for the computation of `rhsz` as the tool advised. Also, following the tool advise, we made a transposition in the `zsolve`. The `exact_rhs` was outside of the timed code section so we did not do any changes in it. The impact of the data traffic optimization on the code performance is shown in Figure 8. The improvement was about 20% for both `rhsz` and `zsolve` and total improvement for the application performance was about 10% on 16 processors.

In the final version of the paper we will present similar data on the other 3 codes: BT, LU and ARC3D.

6. Conclusions and Related Work

We showed on examples that on DSM machines improvements in data traffic result in improvement of the application performance. We demonstrated a few methods for improving data traffic: data placement, data transposition, reduction of thread interference, and reduction of array self and cross interference in cache. The methods can be formalized on the basis of analysis of data-to-computation and data-to-data affinity and can be implemented in a tool advising a user on data traffic improvements. We implemented the methods in as extensions to ADAPT [9].

Data distribution and data locality are fields of intensive research in the last decade. These two aspects of data locality were considered separately: HPF stile of data distributions for distributed memory machines [2][3][9][16][17][18][19][20][21][22] and the spatial and temporal data locality for improving for improving performance of cache based single processor machines [6][7][10][15][25][26][27]. The DSM machines employing ccNUMA architecture exhibit synergetic effects of data distributions and data localization [24]. These effects can range from a superlinear speedup for applications where both aspects of data placement are handled well to a significant slow down if data sharing is handled poor.

Acknowledgments: The authors wish to acknowledge Rob F. Van Der Wijngaart for useful discussions of the cache optimizations and Amy Fornal for providing a library to measure hardware events on the level of nests. The work presented in the paper is supported under NASA High Performance Computing and Communication Program and was executed under NAS task order A61812D.

References

[1] A.V. Aho, R. Sethi and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publ., Reading MA, 1988.

[2] J.M. Anderson, M.S. Lam. *Global Optimizations for Parallelism and Locality on Scalable Parallel Machines.* In Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation (PLDI), Albuquerque, NM, June 23-25, 1993.

[3] E. Ayguade, J. Garcia, U. Kremer. *Tools and Techniques for Automatic Data Layout: A Case Study.* Parallel Computing, v. 24 (1998) pp. 557-578.

[4] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, M. Yarrow. *The NAS Parallel Benchmarks 2.0.* Report NAS-95-020, Dec. 1995. http://www.nas.nasa.gov/Software/NPB/download.

[5] J.W.S. Cassels. *Geometry of Numbers.* Cambridge Univ. Press, 1957 (Ch. VIII).

[6] S. Coleman, K.S. McKinley. *Tile size selection using cache organization and data layout.* In Proc. SIGPLAN'95 Conf. on Programming Language Design and Implementation, June 1995, pp. 279-289.

[7] M.Frumkin, R. F. Van der Wijngaart. *Efficient cache use for stencil operations on structured discretization grids.* Submitted to JACM, see also http://arXiv.org/abs/cs.PF/0007027.

[8] M. Frumkin, J. Yan. *HPF Implementation of ARC3D.* Frontiers'99, February 21-25, 1999, Annapolis, pp. 81-88.

[9] M. Frumkin, J. Yan. *Automatic Data Distribution for CFD Applications on Structured Grids.* The 3rd Annual HPF User Group Meeting, Redondo Beach, CA, August 1-2, 1999, 5 pp. Full version: NAS Technical report NAS-99-012, December 99. 27 pp.

[10] S.Gosh, M.Martonosi, S.Malik. *Cache Miss Equations: An Analytical Representation of Cache Misses.* ACM ICS 1997, pp. 317-324.

[11] H. Jin, J. Yan, M. Frumkin. *Automatic Generation of Directive-Based Parallel Programs for Shared Memory Parallel Systems.* In Proceedings of the International Symposium on High Performance Computing, Tokyo, Japan, October 16-18, 2000.

[12] S.P. Johnson, C.S. Ierotheou, M. Cross. *Automatic Parallel Code Generation on Distributed Memory Systems.* Parallel Computing, V. 22 (1996), pp. 227-258.

[13] S.P. Johnson, M. Cross, M.G. Everett. *Exploitation of Symbolic Information in Interprocedural Dependence Analysis.* Parallel Computing, v. 22 (1966) pp.197-226.

[14] S. Ghosh, M. Martonosi, S. Malik. *Cache Miss Equations: An Analytical Representation of Cache Misses.* ICS 1997, pp. 317-324.

[15] J.L. Hennessy, D.A. Patterson. *Computer Organization and Design.* Morgan Kaufmann Publishers, San Mateo, CA 1994.

[16] K. Kennedy, U. Kremer. *Automatic Data Layout for High Performance Fortran.* Supercomputing '95, San Diego, CA, December 1995.

[17] C.H. Koelbel, D.B. Loverman, R. Shreiber, G.L. Steele Jr., M.E. Zosel. *The High Performance Fortran Handbook.* MIT Press, 1994.

[18] U. Kremer. *Automatic Data Layout for Distributed Memory Machines,* PhD. thesis, Rice Univ., October 1995, CRPC-TR95-599-S.

[19] J. Li, M. Chen. *The Data Alignment Phase in Compiling Programs for Distributed-Memory Machines.* J. of Parallel and Distr. Computing, V. 13, n. 2, August 1991, pp. 213-221.

[20] A.W. Lim, M.S. Lam. *Maximizing Parallelism and Minimizing Synchronization with Affine Partitions.* Parallel Computing, v. 24 (1998), pp. 445-475.

[21] D.J. Palermo, P. Banerjee. *Automatic Selection of Dynamic Data Partitioning Schemes for Distributed-Memory Multicomputers.* In Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing, Columbus, OH, August 1995, LNCS, v. 1033, pp. 392-406, Springer Verlag, 1996.

[22] D.J. Palermo, E.W. Hodges IV, P. Banerjee. *Interprocedural Array Redistribution Data-Flow Analysis.* 9th Workshop on Languages and Compilers for Parallel Computing, San Jose, CA, August 8-10, 1996.

[23] T.H. Pulliam, D.S. Chaussee. *A Diagonal Form of an Implicit Approximate Factorization Algorithm.* Journal of Computational Physics, Vol. 29, p.1037, 1975.

[24] SGI Technical Document. *SGI Origin2000 and Onyx2 Performance Tuning and Optimization Guide.* http://techpubs.sgi.com.

[25] G. Rivera, C.W. Tseng. *Data Transformations for Eliminating Conflict Misses,* PLDI 1998, pp. 38-49.

[26] G. Rivera, C.W. Tseng. *Eliminating Conflict Misses for High Performance Architectures,* ICS 1998, pp. 353-360.

[27] M.E. Wolf, M. Lam. *A Data Locality Optimizing Algorithm.* In Proc. SIGPLAN'91 Conf. on Programming Language Design and Implementation, June 1991, pp. 30-44.

```
do k=1,nz
 do j=1,ny
  do i=1,nx
    q(i,j,k) = u(i,j,k)
              + c1*u(i-2,j,k)+c2*u(i-1,j,k)+c3*u(i+1,j,k)+c4*u(i+2,j,k)
              + c5*u(i,j-2,k)+c6*u(i,j-1,k)+c7*u(i,j+1,k)+c8*u(i,j+2,k)
              + c9*u(i,j,k-2)+c10*u(i,j,k-1)+c11*u(i,j,k+1)+c12*u(i,j,k+2)
   end do
  end do
 end do
```

FIGURE 1. The `kji` nest ordering of the second order explicit operator $q=Ku$.

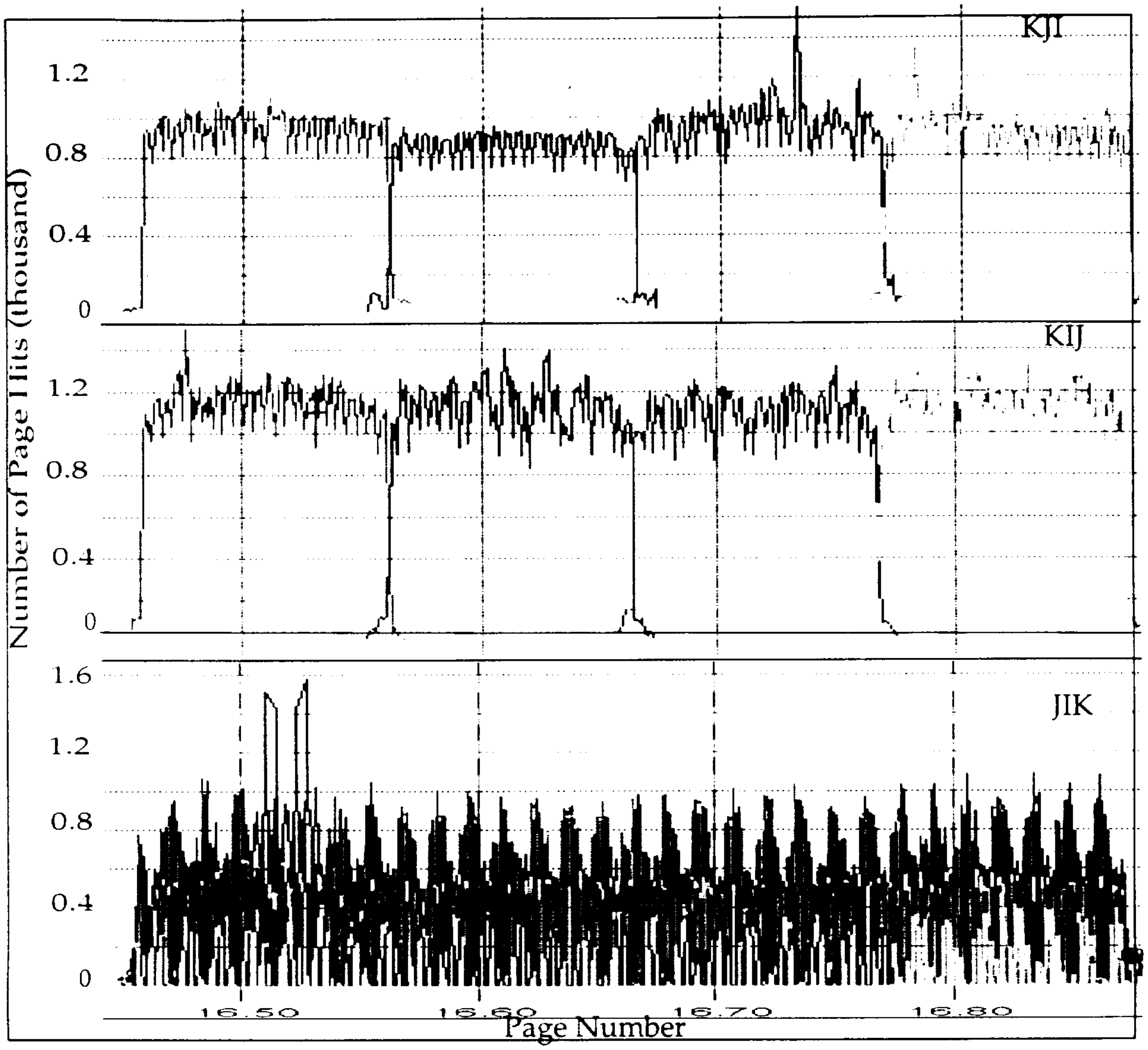

FIGURE 2. Histograms of the memory access addresses obtained with SGI `dprof` tool (OpenMP program with 4 threads).

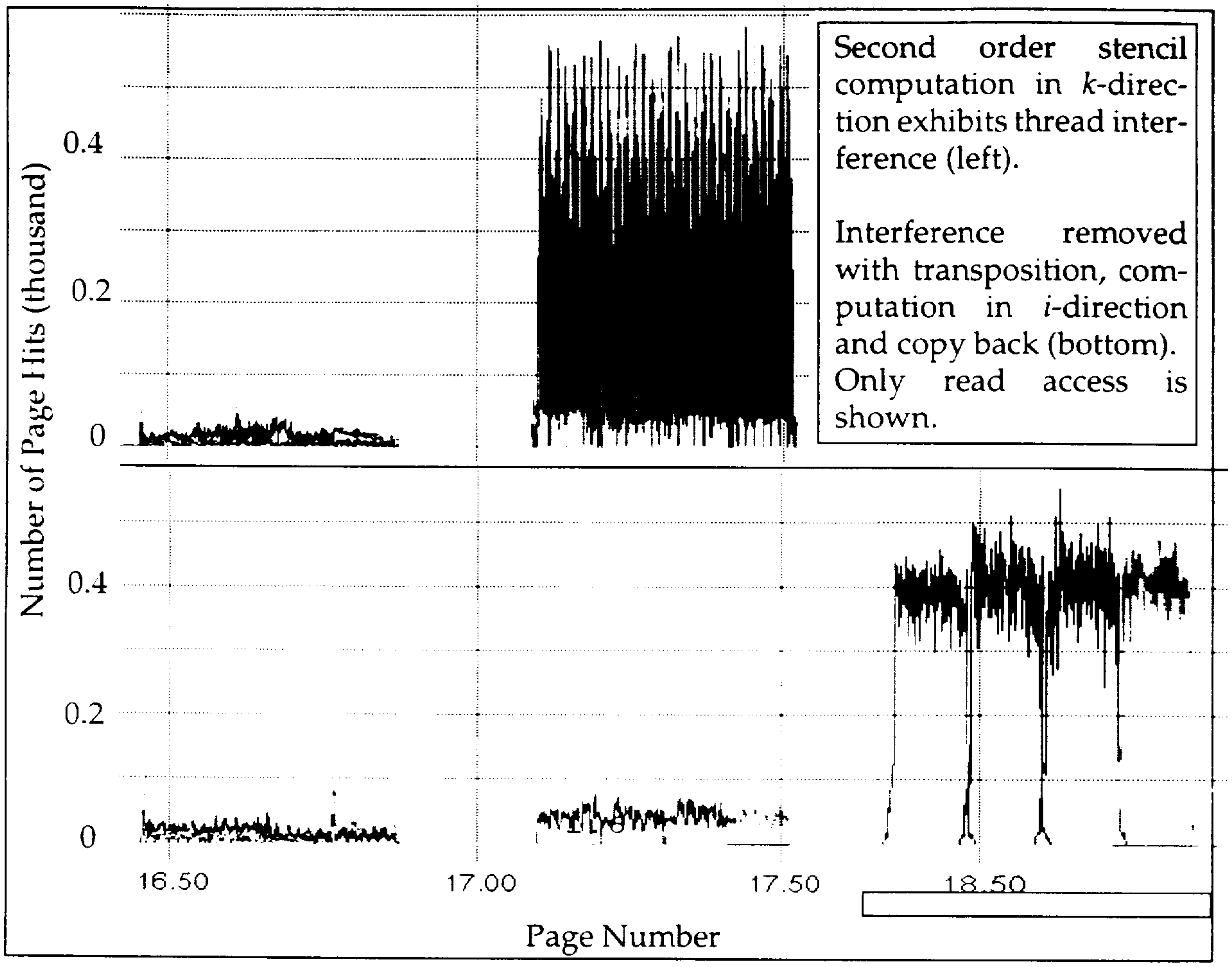

FIGURE 3. Removing memory access interference with transposition. The significant nose in the histograms is a result of sampling of memory addresses and of many fractional pages affine to the threads.

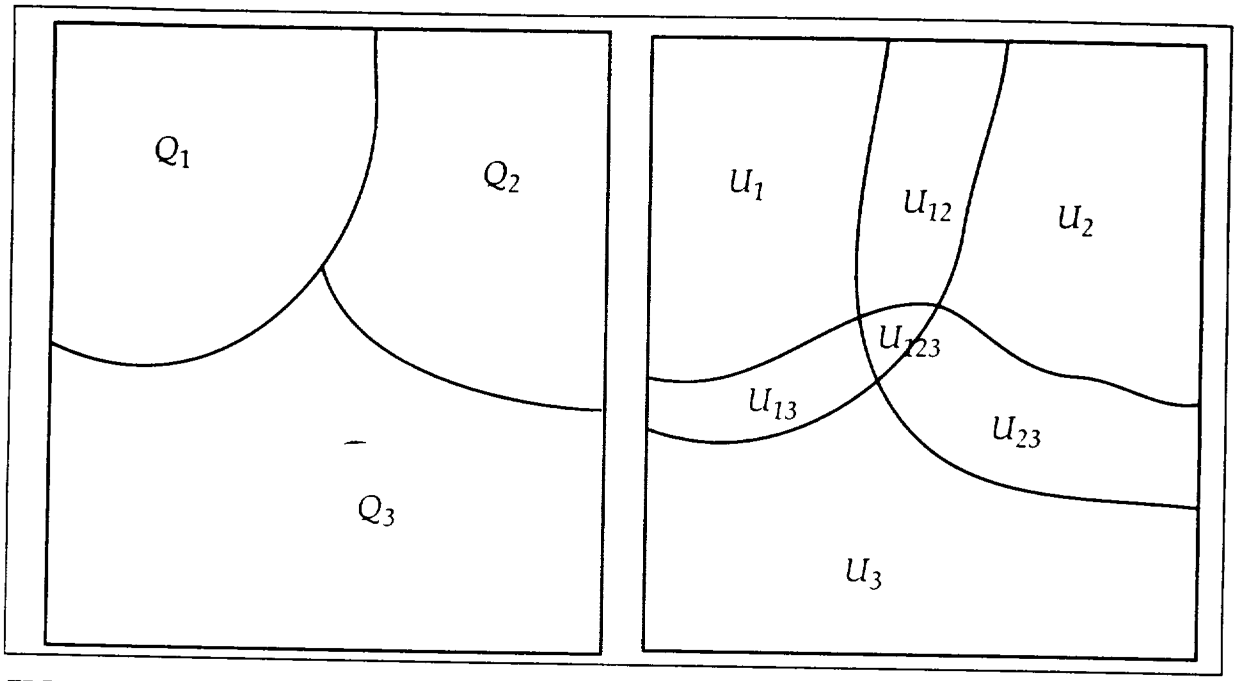

FIGURE 4. A partition of the elements of array q into 3 sets Q_i each computed within time interval $[t_i, t_{i+1}]$, $i=1,2,3$. U_i=aff(Q_i) and U_{ij} is the intersection of U_i and U_j, i,j=1,2,3 (U_{123} is contained in all U_i).

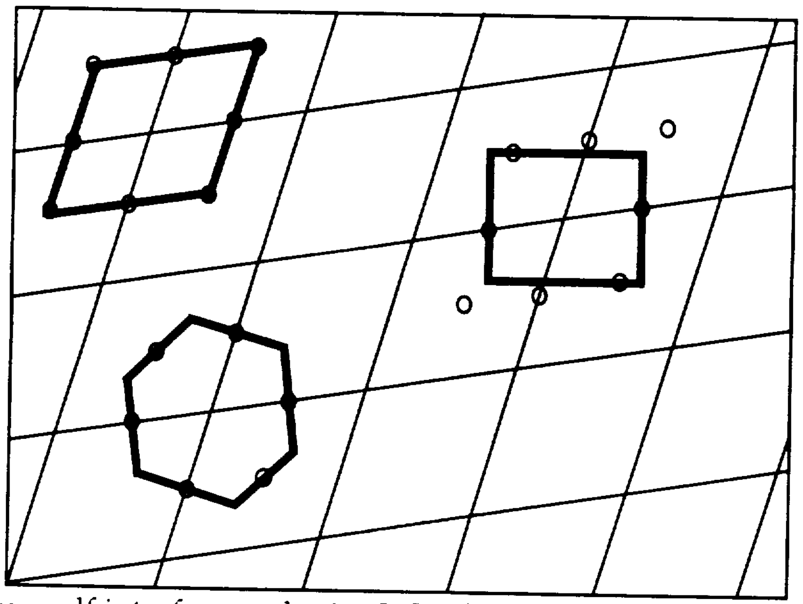

FIGURE 5. Array self interference lattice L. Highlighted are three interference free polygons: a fundamental parallelepiped of the lattice, the Voronoi hexagon, and a rectangular built on the vectors of the successive minima of the lattice $(1/2)L$, see [5]. Each polygon can be used for tiling entire grid.

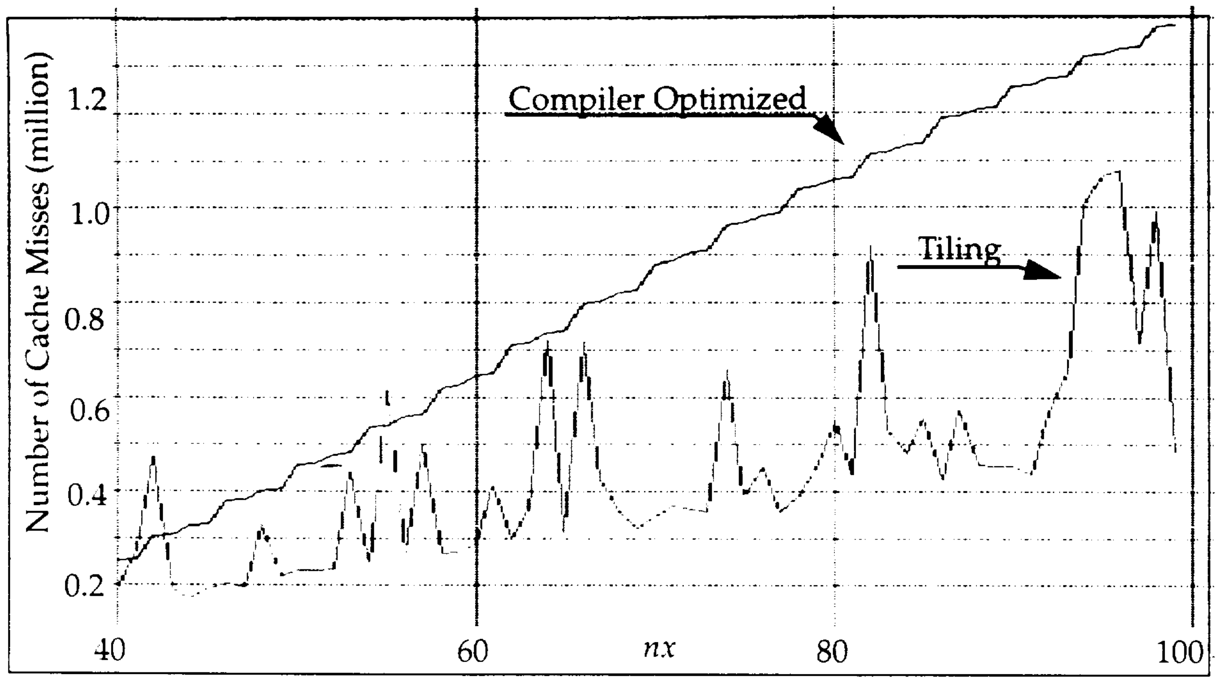

FIGURE 6. Comparison of cache misses for the second order stencil operator as a function of the first dimension (*ny=97, nz=99*). The first graph shows the number of cache misses for compiler optimized nest. The second graph is obtained for tiling with successive minima parallelepiped.

lhsz

```
do j=1,ny
  do i=1,nx
    do k=1,nz
      cv(k) = ws(i,j,k)
      rhon(k) = SimpleFunction(rho_i(i,j,k))
    end do
    do k=1,nz
      lhs(i,j,k,1) =   0.0d0
      lhs(i,j,k,2) = - dttx2 * cv(k-1) - dttx1 * rhon(k-1)
      lhs(i,j,k,3) =   1.0d0 + c2dttx1 * rhon(k)
      lhs(i,j,k,4) =   dttx2 * cv(k+1) - dttx1 * rhon(k+1)
      lhs(i,j,k,5) =   0.0d0
    end do
  end do  —
end do
```

lhsz_aff

```
(j,i,k;lhs(i,j,k,1))
(j,i,k;lhs(i,j,k,2),ws(i,j,k-1),rho_i(i,j,k-1))
(j,i,k;lhs(i,j,k,3),rho_i(i,j,k))
(j,i,k;lhs(i,j,k,4),ws(i,j,k+1),rho_i(i,j,k+1))
(j,i,k;lhs(i,j,k,5))
```

lhsz_r

```
do k=1,nz
  do j=1,ny
    do i=1,nx
      lhs(i,j,k,1) =  0.0d0
      lhs(i,j,k,2) = -dttz2*ws(i,j,k-1)
                     -dttz1*SimpleFunction(rho_i(i,j,k-1))
      lhs(i,j,k,3) =  1.0 + c2dttz1*SimpleFunction(rho_i(i,j,k))
      lhs(i,j,k,4) =  dttz2*ws(i,j,k+1)
                     -dttz1*SimpleFunction(rho_i(i,j,k+1))
      lhs(i,j,k,5) =  0.0d0
    end do
  end do
end do
```

lhsz_r_aff

```
(k,j,i;lhs(i,j,k,1))
(k,j,i;lhs(i,j,k,2),ws(i,j,k-1),rho_i(i,j,k-1))
(k,j,i;lhs(i,j,k,3),rho_i(i,j,k))
(k,j,i;lhs(i,j,k,4),ws(i,j,k+1),rho_i(i,j,k+1))
(k,j,i;lhs(i,j,k,5))
```

FIGURE 7. A nest transformation to reduce TLB misses and PDC misses. The affinity relation `lhsz_aff` for nonprivatizable arrays of the `lhsz` suggests a large stride in memory accesses and little reuse of data in cache. `lhs_r_aff` has a reverse index mapping as evidence of a good memory access pattern. The index of the parallelized loop is shown in bold.

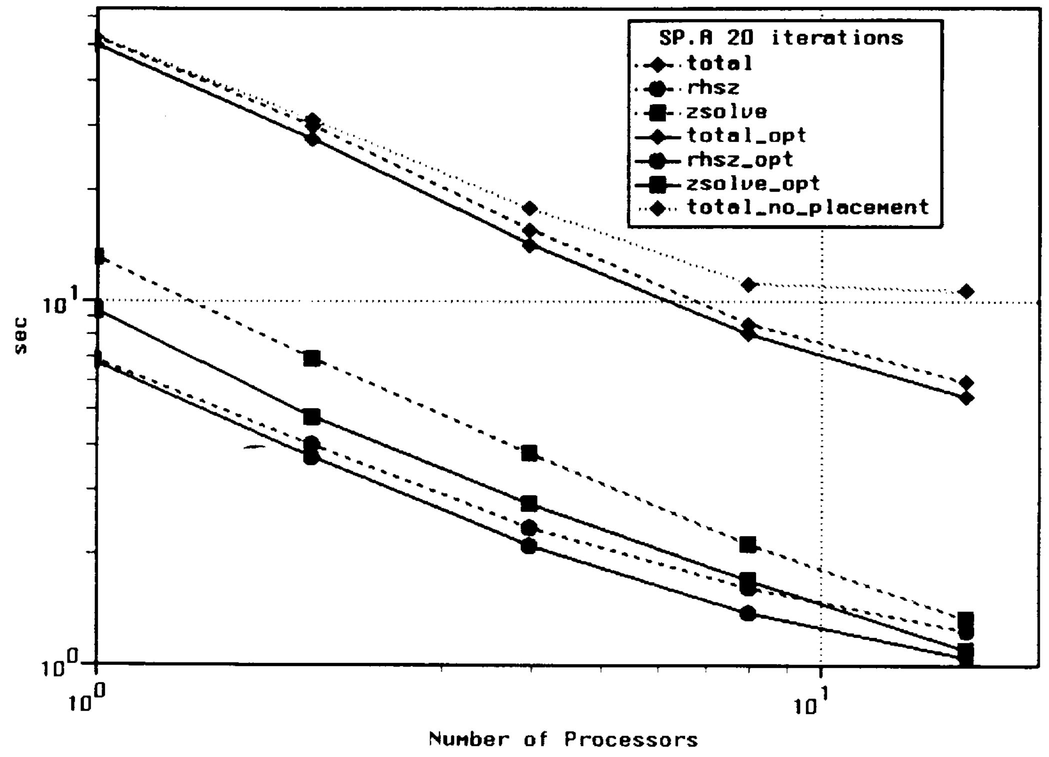

FIGURE 8. The impact of data traffic optimization on the execution time of OpenMP version of NAS Parallel Benchmark SP class A [4]. The top curve total_no_placement shows the effect of a concentration of arrays in a single node memory. The dashed curves show performance of the original code, and solid curves show performance of the optimized code.

TABLE 1. Profiling PDC, SDC, and TLB misses for the second order stencil computations with SGI hardware counters. Data are presented for the master thread, time is given in microseconds. The page size=16k.

	1 thread				4 threads			
nest type	**TIME**	**PDC**	**SDC**	**TLB**	**TIME**	**PDC**	**SDC**	**TLB**
KJI	124	1219645	106130	569	30	304964	758	145
KIJ	162	4760811	106886	552	39	1190243	1214	159
JIK	631	2478688	104002	1648271	167	639880	4471	442474
JIK+ transpose	333	1267223	111342	703828	91	327166	19836	181840
KJI + data placement	124	1219675	106130	554	25	304960	916	142

TABLE 2. Profiling PDC, SDC, and TLB misses for the second order stencil computations with SGI hardware counters. Data are presented for the master thread, time is given in microseconds. The page size=1M.

	1 thread				4 threads			
nest type	**TIME**	**PDC**	**SDC**	**TLB**	**TIME**	**PDC**	**SDC**	**TLB**
KJI	147	1304209	112001	5	30	305020	819	0
KIJ	183	4800928	112034	7	39	1190798	1283	1
JIK	175	4771958	101661	4	64	499695	4748	14
JIK+ transpose	145	1467570	112348	0	45	310805	19822	24
KJI + data placement	145	1304479	111977	5	26	305032	804	0

TABLE 3. Measurements of hardware events in the first nest of `lhsx`, `lhsy`, and `lhsz`. The second two columns show measurements for the rearranged nests.

Counter name \ nest	**lhsx**	**lhsy**	**lhsz**	**lhsy_r**	**lhsz_r**
Graduated FP instructions	5842922	5842916	5842928	12869754	12869748
TLB misses	1157	1274	3487850	1169	2261
PDC misses	876941	2172573	2183645	884053	1251526
SDC misses	216528	221182	221438	221234	221952
Execution time (ms)	151	222	1372	175	191